Clordane
An interactive debugger for
Common Lisp

Robert Strandh

2015

Contents

Chapter 1

Introduction

1.1 Purpose

Clordane is a debugger for Common Lisp. The Common Lisp standard has an entry for the word "debugger" in the glossary, but there are no requirements on the capabilities of a debugger there, other than that it should allow the user to handle conditions interactively. Most existing implementations provide more functionality, such as the ability to examine active stack frames including the values of live variables and the source location of the return address of a particular stack frame.

In this document, we define a debugger with many more features than what is typically provided. In particular, we take advantage of the existence of *threads* in most modern Common Lisp implementations to define improvements to the operations provided by the debugger.

The downside of these additional features is that Clordane will require much more assistance from the supported implementations than required by existing debuggers.

1.2 Terminology

1.2.1 Application thread

From the point of view of Clordane, the application thread is the thread that the programmer wants to debug. Other threads running the same code as the application thread are not affected by the debugging actions taken by the programmer.

The application thread can run any code, including that of Clordane.

1.2.2 Debugger thread

The debugger thread is the thread running Clordane. The debugger thread can not debug itself, but it can debug the same code running in a different thread.

1.2.3 Poll point

A *poll point* is a place in the program where a running thread can be *stopped*.

From the point of view of the compiler, a poll point is a place where code is inserted so that the program interrogates its *thread* to see whether it should take some action, such as generating trace output, stopping the execution, or some other action. For a possible implementation of poll points, see Appendix A.

The compiler generates poll points only where it is *safe* to stop the program.

Compiling code with a higher `debug` value gives executable code with more poll points.

1.2.4 Break point

A *break point* is a poll point that has been marked to indicate that the running program should stop its execution when it is reached.

Such break points can be *created* or *destroyed* for a thread by Clordane independently of whether the thread is currently executing or currently stopped.

A break point can be *steady* or *volatile*. A *steady* break point is destroyed only as a result of an action on the part of the programmer. All *volatile* break points are destroyed when the program stops at any breakpoint.

A break point can be associated with a Common Lisp *form* to be evaluated when the break point is reached. The form is evaluated by Clordane. Tracing can be accomplished by associating with the break point a form that will print some information and then *continue* the execution of the thread.

1.2.5 Stopping point

A *stopping point* is a place in the program where the thread of execution is currently stopped. A stopping point can be any program counter value, but a value of the program counter other than a poll point can not be reached by creating a break point. It can, however, be reached by *stepping by instruction* after the program has stopped at a poll point.

1.2.6 Continue

One action the programmer can take when the application thread is stopped is to instruct Clordane to *continue* the execution of the application thread. The execution of the application thread will then resume until it either terminates or reaches a break point.

1.2.7 Advance

When the application thread is stopped, the application programmer can instruct Clordane to *advance* to a particular poll point. Clordane will then insert a *volatile break point* at that point and then continue the execution of the application thread.

1.2.8 Step

The application programmer can instruct Clordane to *step* the execution of the application thread, either by *poll point* or by *instruction*.

Stepping by poll point comes in three variants, namely *step in*, *step out*, and *step over*. The *step in* command steps to the next poll point *inside* the next form to be evaluated. The *step out* command finishes the execution of the current form and stops at the end of that form. The *step over* command evaluates the next form and then stops at the end if it, except when there is more than one possible next form (as a result of a condition) when it stops at the *beginning* of the next form.

When Clordane is instructed to step by poll point, it inserts one or more *volatile break points* at appropriate poll points in the program, and then continues executions until some break point is reached, at which point it will *stop* the application thread and destroy all volatile break points independently of the kind of break point that was reached.

When Clordane is instructed to step by instruction, it will execute the next machine instruction and then stop the execution of the application thread.

Chapter 2

Different types of windows

2.1 Source window

A source window is a window that displays source code together with poll points. A poll point which is not a break point is shown as a cursor with blue color. A poll point which is also a break point is shown as a cursor with red color.

2.2 Stopping point window

A stopping point window is a window that is popped up whenever the application thread is stopped.

The stopping point window shows the position of the stopping point and variables that are *live* at the stopping point. The value of a live variable can be obtained by hovering the pointer over the variable.

In the stopping point window is also shown poll points in the source code to which the user can ask the program to *advance*. Clicking with the left mouse button on such a point will set a temporary stopping point at that position and then the program will continue from the stopping point. The temporary stopping point is removed as soon as it is reached.

In this window, there are button that the user can click on. One such button is marked *Finish* and results in the program continuing from the stopping point to immediately after the next *return*.

Another button is marked *Enter*. This button is clickable only if the stopping point immediately precedes a function call or a tail call. Clicking this button sets a temporary stopping point at the start of the function about to be called and then the program execution continues from the stopping point.

2.3 Stack backtrace window

Whenever the program is stopped, the *stack backtrace window* is updated. See Chapter 6 for more information on the backtrace facility.

2.4 Interaction window

This window provides the user with a `read-eval-print` loop, augmented with the possibility of submitting *commands* by starting an interaction with the comma character.

Chapter 3

Breakpoints

3.1 Creating or destroying a breakpoint

When the mouse is in a source window, poll point are shown as blue cursors. A break point set are shown as red cursors.

Clicking with the left mouse button on a blue cursor will create a break point at that poll point, and the cursor will change color to red.

Clicking with the left mouse button on a red cursor will pop up a menu with several choices concerning that break point:

- Destroy the breakpoint.
- Add a form. This choice will open an code input window allowing the user to type a form. Each time the program control reaches a break point, the form is evaluated.
- Remove the form.
- Etc.

Chapter 4

Examining and modifying data

When the application is stopped at some stopping point, Clordane provides a REPL for the user. This REPL is not the ordinary REPL of the Common Lisp implementation, in that it operates in the lexical environment of the stopping point. Also, the available functionality of the REPL may be restricted so as to preserve the integrity of the system.

In particular, a references to a lexical variable is resolved to the place where the compiler allocated space for it, which might be in a register or on the stack. The set of available such references may depend on the `debug` level when the code was compiled. The application programmer may use `setq` to alter the value of lexical variables. The value to be assigned is then restricted to the *type* that the compiler inferred for this variable so that the code behaves in a consistent way after an assignment. Implementations may of course choose to disable type inference so that any value is allowed, or they may disallow references to lexical variables for which they have no Clordane assignment support.

In addition to providing a REPL, Clordane automatically displays all available lexical variables at the stopping point, allowing the application programmer to see which variables can be referred to. Each variable is shown with its name, its value (which may be clicked on for inspection), and its type.

Chapter 5

Stepping

Stepping refers to the action of making the execution of the debugged program advance by a specific unit of code. There are several ways for the programmer to determine the unit of code to step by:

- If the current stopping point is located immediately *before* some expression, then the step may execute that expression and then stop when that expression is completely evaluated. This action is called *step over*.
- If the current stopping point is located immediately *before* some expression, then the step may execute that expression and then stop immediately before the next expression to be evaluated. This action is called *step next*
- If the current stopping point is located immediately *after* some expression, then the step may result in no execution, and then stop immediately before the next expression to be evaluated. This action is called *step null*
- If the current stopping point is located immediately *after* some expression, then the step may execute the next expression to be evaluated and then stop immediately after that expression. This action is called *step next* and is distinguished from the previous action by the initial position of the stopping point. If the stopping point is both immediately after some expression and immediately before the following expression, then this action is equivalent to *step over*.

- If the current stopping point is located immediately *before* some expression and that expression is a function call, then the step may be to enter the called function and stop before the first expression of that function is evaluated. This action is called *step in*
- The remaining expressions of the currently executing function may be evaluated and execution stopped immediately after that function returns to its caller. This action is called *step finish*.
- The step could be a single machine instruction in which case the next instruction is executed and then the execution is stopped. This action is called *step instruction*.

Chapter 6

Backtrace

When an unhandled error is signaled in the application thread, the execution of that thread is stopped, and Clordane displays a *backtrace*. By default, the backtrace shows only function calls that are part of the application itself, and not part of the internal workings of the Common Lisp system. A button can be clicked on to toggle this setting so that all stack frames are shown.

Clicking on one of the stack frames makes this stack frame the *current* one. A different stack frame can also be selected by navigating with the keystrokes `p` (for *previous*) and `n` (for *next*). The Clordane REPL can then be used to evaluate forms in the lexical environment of that stack frame, and the `*package*` special variable is set to the package in which the code of the current stack frame has been defined.

Initially, the *stoppingpoint window* discussed in Section 2.2 shows the point in the code where the error was signaled. When the user selects some stack frame, the stoppingpoint window shows the code around the point where execution will resume if control is returned to that stack frame. As before, the value of live variables can be obtained by hovering the pointer over a variable.

Appendix A

Implementing poll points

There are many ways of implementing poll points. However, we consider that a reasonable implementation must have the following characteristics:

- There should be little or no performance penalty for code that is running in some thread other than a thread being debugged.
- There should be little performance penalty for poll points that are not currently breakpoints.

We suggest the following technique which will work for any processor architecture:

- The thread instance contains three pieces of information associated with poll points:
  1. A *debug flag* which is a single bit indicating whether the thread is running under the control of the debugger. At the beginning of each top-level function compiled with a high value of the `debug` quality, this flag is loaded into a lexical variable, subject to register allocation as usual.
  2. A *breakpoint table* (perhaps a hash table) indexed by values of the program counter, and containing information about the nature of a breakpoint at this program point.

3. A small *bit table*, say 1024 bits or so. The index into this table is the value of the program counter modulo its size. It contains summary information from the previous table, i.e., whenever some value of the program counter is present in the breakpoint table, then the corresponding bit is set in this bit table.

- When code is compiled with a high value of the `debug` quality, a small, local routine is added to each top-level function. This routine is called using an inexpensive call protocol, typically just a `jsr` instruction. The routine performs the following actions:
  - It starts by checking the debug flag that was loaded into a lexical variable from the thread instance of the running thread. If that flag is cleared, then the routine returns without any action.
  - If the flag is set, then it uses the value of the program counter stored on the stack as part of the call protocol. It takes that value modulo the size of the bit table and checks whether the entry in the bit table is set. If it is cleared, again, the routine returns without any action.
  - If the bit-table entry is set, then it consults the breakpoint table to see whether the actual value of the program counter is present there. If not, the routine again returns normally. If it is present, then it suspends the execution of the thread and hands over control to the debugger thread.
- Also when code is compiled with a high value of the `debug` quality, a call to the small routine is inserted before the beginning, and after the end of the execution of every source-level form present in the code.

The debugger maintains the breakpoint table and the bit table as follows:

- When a breakpoint is set, it is added to the breakpoint table, and bit in the bit table corresponding to the value of the program counter modulo the size of the bit table is set.
- When a breakpoint is removed, the entire bit table is first cleared. Then the breakpoint table is traversed and, for each breakpoint, the corresponding bit in the bit table is set as before.

Bibliography